15 Inductive and Coinductive Types

The *inductive* and the *coinductive* types are two important forms of recursive type. Inductive types correspond to *least*, or *initial*, solutions of certain type equations, and coinductive types correspond to their *greatest*, or *final*, solutions. Intuitively, the elements of an inductive type are those that are given by a finite composition of its introduction forms. Consequently, if we specify the behavior of a function on each of the introduction forms of an inductive type, then its behavior is defined for all values of that type. Such a function is a *recursor*, or *catamorphism*. Dually, the elements of a coinductive type are those that behave properly in response to a finite composition of its elimination forms. Consequently, if we specify the behavior of an element on each elimination form, then we have fully specified a value of that type. Such an element is a *generator*, or *anamorphism*.

15.1 Motivating Examples

The most important example of an inductive type is the type of natural numbers as formalized in Chapter 9. The type $\mathtt{nat}$ is the *least* type containing $\mathtt{z}$ and closed under $\mathtt{s}(-)$. The minimality condition is expressed by the existence of the iterator, $\mathtt{iter}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$, which transforms a natural number into a value of type τ, given its value for zero, and a transformation from its value on a number to its value on the successor of that number. This operation is well-defined precisely because there are no other natural numbers.

With a view towards deriving the type $\mathtt{nat}$ as a special case of an inductive type, it is useful to combine zero and successor into a single introduction form, and to correspondingly combine the basis and inductive step of the iterator. The following rules specify the statics of this reformulation:

$$\frac{\Gamma \vdash e : \mathtt{unit} + \mathtt{nat}}{\Gamma \vdash \mathtt{fold}_{\mathtt{nat}}(e) : \mathtt{nat}} \tag{15.1a}$$

$$\frac{\Gamma, x : \mathtt{unit} + \tau \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2) : \tau} \tag{15.1b}$$

The expression $\mathtt{fold}_{\mathtt{nat}}(e)$ is the unique introduction form of the type $\mathtt{nat}$. Using this, the expression $\mathtt{z}$ is $\mathtt{fold}_{\mathtt{nat}}(\mathtt{l} \cdot \langle\rangle)$, and $\mathtt{s}(e)$ is $\mathtt{fold}_{\mathtt{nat}}(\mathtt{r} \cdot e)$. The recursor, $\mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2)$, takes as argument the abstractor $x.e_1$ that combines the basis and inductive step into a single computation that, given a value of type $\mathtt{unit} + \tau$, yields a value of type τ. Intuitively, if

https://doi.org/10.1017/CBO9781316576892.017 Published online by Cambridge University Press

x is replaced by the value $\mathtt{l} \cdot \langle\rangle$, then e_1 computes the base case of the recursion, and if x is replaced by the value $\mathtt{r} \cdot e$, then e_1 computes the inductive step from the result e of the recursive call.

The dynamics of the combined representation of natural numbers is given by the following rules:

$$\frac{}{\mathtt{fold}_{\mathtt{nat}}(e)\ \mathsf{val}} \tag{15.2a}$$

$$\frac{e_2 \longmapsto e_2'}{\mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2) \longmapsto \mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2')} \tag{15.2b}$$

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}(x.e_1; \mathtt{fold}_{\mathtt{nat}}(e_2)) \\ \longmapsto \\ [\mathtt{map}\{t.\mathtt{unit} + t\}(y.\mathtt{rec}_{\mathtt{nat}}(x.e_1; y))(e_2)/x]e_1\end{array}} \tag{15.2c}$$

Rule (15.2c) uses (polynomial) generic extension (see Chapter 14) to apply the recursor to the predecessor, if any, of a natural number. If we expand the definition of the generic extension in place, we obtain this rule:

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}(x.e_1; \mathtt{fold}_{\mathtt{nat}}(e_2)) \\ \longmapsto \\ [\mathtt{case}\, e_2\, \{\mathtt{l} \cdot \_ \hookrightarrow \mathtt{l} \cdot \langle\rangle \mid \mathtt{r} \cdot y \hookrightarrow \mathtt{r} \cdot \mathtt{rec}_{\mathtt{nat}}(x.e_1; y)\}/x]e_1\end{array}}$$

Exercise **15.2** asks for a derivation of the iterator, as defined in Chapter 9, from the recursor just given.

An illustrative example of a coinductive type is the type of *streams* of natural numbers. A stream is an infinite sequence of natural numbers such that an element of the stream can be computed only after computing all preceding elements in that stream. That is, the computations of successive elements of the stream are sequentially dependent in that the computation of one element influences the computation of the next. In this sense, the introduction form for streams is dual to the elimination form for natural numbers.

A stream is given by its behavior under the elimination forms for the stream type: $\mathtt{hd}(e)$ returns the next, or head, element of the stream, and $\mathtt{tl}(e)$ returns the tail of the stream, the stream resulting when the head element is removed. A stream is introduced by a *generator*, the dual of a recursor, that defines the head and the tail of the stream in terms of the current state of the stream, which is represented by a value of some type. The statics of streams is given by the following rules:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{hd}(e) : \mathtt{nat}} \tag{15.3a}$$

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{tl}(e) : \mathtt{stream}} \tag{15.3b}$$

https://doi.org/10.1017/CBO9781316576892.017 Published online by Cambridge University Press

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \mathtt{nat} \quad \Gamma, x : \tau \vdash e_2 : \tau}{\Gamma \vdash \mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2> : \mathtt{stream}} \tag{15.3c}$$

In rule (15.3c), the current state of the stream is given by the expression e of some type τ, and the head and tail of the stream are determined by the expressions e_1 and e_2, respectively, as a function of the current state. (The notation for the generator is chosen to emphasize that every stream has both a head *and* a tail.)

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2> \mathsf{val}} \tag{15.4a}$$

$$\frac{e \longmapsto e'}{\mathtt{hd}(e) \longmapsto \mathtt{hd}(e')} \tag{15.4b}$$

$$\frac{}{\mathtt{hd}(\mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2>) \longmapsto [e/x]e_1} \tag{15.4c}$$

$$\frac{e \longmapsto e'}{\mathtt{tl}(e) \longmapsto \mathtt{tl}(e')} \tag{15.4d}$$

$$\frac{}{\begin{array}{c}\mathtt{tl}(\mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2>) \\ \longmapsto \\ \mathtt{strgen}\, x\, \mathtt{is}\, [e/x]e_2\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2>\end{array}} \tag{15.4e}$$

Rules (15.4c) and (15.4e) express the dependency of the head and tail of the stream on its current state. Observe that the tail is obtained by applying the generator to the new state determined by e_2 from the current state.

To derive streams as a special case of a coinductive type, we combine the head and the tail into a single elimination form, and reorganize the generator correspondingly. Thus, we consider the following statics:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{unfold}_{\mathtt{stream}}(e) : \mathtt{nat} \times \mathtt{stream}} \tag{15.5a}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \mathtt{nat} \times \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{gen}_{\mathtt{stream}}(x.e_1; e_2) : \mathtt{stream}} \tag{15.5b}$$

Rule (15.5a) states that a stream may be unfolded into a pair consisting of its head, a natural number, and its tail, another stream. The head $\mathtt{hd}(e)$ and tail $\mathtt{tl}(e)$ of a stream e are the projections $\mathtt{unfold}_{\mathtt{stream}}(e) \cdot \mathtt{l}$ and $\mathtt{unfold}_{\mathtt{stream}}(e) \cdot \mathtt{r}$, respectively. Rule (15.5b) states that a stream is generated from the state element e_2 by an expression e_1 that yields the head element and the next state as a function of the current state.

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{gen}_{\mathtt{stream}}(x.e_1; e_2)\ \mathsf{val}} \tag{15.6a}$$

https://doi.org/10.1017/CBO9781316576892.017 Published online by Cambridge University Press

$$\frac{e \longmapsto e'}{\mathtt{unfold}_{\mathtt{stream}}(e) \longmapsto \mathtt{unfold}_{\mathtt{stream}}(e')} \tag{15.6b}$$

$$\frac{}{\begin{array}{c}\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}(x.e_1; e_2)) \\ \longmapsto \\ \mathtt{map}\{t.\mathtt{nat} \times t\}(y.\mathtt{gen}_{\mathtt{stream}}(x.e_1; y))([e_2/x]e_1)\end{array}} \tag{15.6c}$$

Rule (15.6c) uses generic extension to generate a new stream whose state is the second component of $[e_2/x]e_1$. Expanding the generic extension we obtain the following reformulation of this rule:

$$\frac{}{\begin{array}{c}\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}(x.e_1; e_2)) \\ \longmapsto \\ \langle([e_2/x]e_1) \cdot \mathtt{l}, \mathtt{gen}_{\mathtt{stream}}(x.e_1; ([e_2/x]e_1) \cdot \mathtt{r})\rangle\end{array}}$$

Exercise **15.3** asks for a derivation of $\mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2>$ from the coinductive generation form.

15.2 Statics

We may now give a general account of inductive and coinductive types, which are defined in terms of positive type operators. We will consider a variant of **T**, which we will call **M**, with natural numbers replaced by functions, products, sums, and a rich class of inductive and coinductive types.

15.2.1 Types

The syntax of inductive and coinductive types involves *type variables*, which are, of course, variables ranging over types. The abstract syntax of inductive and coinductive types is given by the following grammar:

Typ	τ	$::=$	t	t	self-reference
			$\mathtt{ind}(t.\tau)$	$\mu(t.\tau)$	inductive
			$\mathtt{coi}(t.\tau)$	$\nu(t.\tau)$	coinductive

Type formation judgments have the form

$$t_1\ \mathsf{type}, \ldots, t_n\ \mathsf{type} \vdash \tau\ \mathsf{type},$$

where $t_1, \ldots, t_n$ are type names. We let Δ range over finite sets of hypotheses of the form $t\ \mathsf{type}$, where t is a type name. The type formation judgment is inductively defined by the

https://doi.org/10.1017/CBO9781316576892.017 Published online by Cambridge University Press

following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{15.7a}$$

$$\frac{}{\Delta \vdash \mathtt{unit}\ \mathsf{type}} \tag{15.7b}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{prod}(\tau_1; \tau_2)\ \mathsf{type}} \tag{15.7c}$$

$$\frac{}{\Delta \vdash \mathtt{void}\ \mathsf{type}} \tag{15.7d}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{sum}(\tau_1; \tau_2)\ \mathsf{type}} \tag{15.7e}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1; \tau_2)\ \mathsf{type}} \tag{15.7f}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta \vdash t.\tau\ \mathsf{pos}}{\Delta \vdash \mathtt{ind}(t.\tau)\ \mathsf{type}} \tag{15.7g}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta \vdash t.\tau\ \mathsf{pos}}{\Delta \vdash \mathtt{coi}(t.\tau)\ \mathsf{type}} \tag{15.7h}$$

15.2.2 Expressions

The abstract syntax of **M** is given by the following grammar:

Exp	e	::=	$\mathtt{fold}\{t.\tau\}(e)$	$\mathtt{fold}_{t.\tau}(e)$	constructor
			$\mathtt{rec}\{t.\tau\}(x.e_1; e_2)$	$\mathtt{rec}(x.e_1; e_2)$	recursor
			$\mathtt{unfold}\{t.\tau\}(e)$	$\mathtt{unfold}_{t.\tau}(e)$	destructor
			$\mathtt{gen}\{t.\tau\}(x.e_1; e_2)$	$\mathtt{gen}(x.e_1; e_2)$	generator

The subscripts on the concrete syntax forms are often omitted when they are clear from context.

The statics for **M** is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\mathtt{ind}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}\{t.\tau\}(e) : \mathtt{ind}(t.\tau)} \tag{15.8a}$$

$$\frac{\Gamma, x : [\tau'/t]\tau \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \mathtt{ind}(t.\tau)}{\Gamma \vdash \mathtt{rec}\{t.\tau\}(x.e_1; e_2) : \tau'} \tag{15.8b}$$

$$\frac{\Gamma \vdash e : \mathtt{coi}(t.\tau)}{\Gamma \vdash \mathtt{unfold}\{t.\tau\}(e) : [\mathtt{coi}(t.\tau)/t]\tau} \tag{15.8c}$$

$$\frac{\Gamma \vdash e_2 : \tau_2 \quad \Gamma, x : \tau_2 \vdash e_1 : [\tau_2/t]\tau}{\Gamma \vdash \mathtt{gen}\{t.\tau\}(x.e_1; e_2) : \mathtt{coi}(t.\tau)} \tag{15.8d}$$

https://doi.org/10.1017/CBO9781316576892.017 Published online by Cambridge University Press

15.3 Dynamics

The dynamics of **M** is given in terms of the positive generic extension operation described in Chapter 14. The following rules specify a lazy dynamics for **M**:

$$\frac{}{\mathtt{fold}\{t.\tau\}(e)\ \mathsf{val}} \tag{15.9a}$$

$$\frac{e_2 \longmapsto e_2'}{\mathtt{rec}\{t.\tau\}(x.e_1; e_2) \longmapsto \mathtt{rec}\{t.\tau\}(x.e_1; e_2')} \tag{15.9b}$$

$$\frac{}{\begin{array}{c}\mathtt{rec}\{t.\tau\}(x.e_1; \mathtt{fold}\{t.\tau\}(e_2)) \\ \longmapsto \\ [\mathtt{map}^+\{t.\tau\}(y.\mathtt{rec}\{t.\tau\}(x.e_1; y))(e_2)/x]e_1\end{array}} \tag{15.9c}$$

$$\frac{}{\mathtt{gen}\{t.\tau\}(x.e_1; e_2)\ \mathsf{val}} \tag{15.9d}$$

$$\frac{e \longmapsto e'}{\mathtt{unfold}\{t.\tau\}(e) \longmapsto \mathtt{unfold}\{t.\tau\}(e')} \tag{15.9e}$$

$$\frac{}{\begin{array}{c}\mathtt{unfold}\{t.\tau\}(\mathtt{gen}\{t.\tau\}(x.e_1; e_2)) \\ \longmapsto \\ \mathtt{map}^+\{t.\tau\}(y.\mathtt{gen}\{t.\tau\}(x.e_1; y))([e_2/x]e_1)\end{array}} \tag{15.9f}$$

Rule (15.9c) states that to evaluate the recursor on a value of recursive type, we inductively apply the recursor as guided by the type operator to the value, and then apply the inductive step to the result. Rule (15.9f) is simply the dual of this rule for coinductive types.

Lemma 15.1. *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

Proof By rule induction on rules (15.9). □

Lemma 15.2. *If $e : \tau$, then either e val or there exists e' such that $e \longmapsto e'$.*

Proof By rule induction on rules (15.8). □

Although a proof of this fact lies beyond our current reach, all programs in **M** terminate.

Theorem 15.3 (Termination for **M**)**.** *If $e : \tau$, then there exists e' val such that $e \longmapsto^* e'$.*

https://doi.org/10.1017/CBO9781316576892.017 Published online by Cambridge University Press

It may, at first, seem surprising that a language with infinite data structures, such as streams, can enjoy such a termination property. But bear in mind that infinite data structures, such as streams, are represented as in a continuing state of creation, and not as a completed whole.

15.4 Solving Type Equations

For a positive type operator $t.\tau$, we may say that the inductive type $\mu(t.\tau)$ and the coinductive type $\nu(t.\tau)$ are both *solutions* (up to isomorphism) of the type equation $t \cong \tau$:

$$\mu(t.\tau) \cong [\mu(t.\tau)/t]\tau$$
$$\nu(t.\tau) \cong [\nu(t.\tau)/t]\tau.$$

Intuitively speaking, this means that every value of an inductive type is the folding of a value of the unfolding of the inductive type, and that, similarly, every value of the unfolding of a coinductive type is the unfolding of a value of the coinductive type itself. It is a good exercise to define functions back and forth between the isomorphic types and to convince yourself informally that they are mutually inverse to one another.

Whereas both are solutions to the same type equation, they are not isomorphic to each other. To see why, consider the inductive type $\texttt{nat} \triangleq \mu(t.\texttt{unit} + t)$ and the coinductive type $\texttt{conat} \triangleq \nu(t.\texttt{unit} + t)$. Informally, `nat` is the smallest (most restrictive) type containing zero, given by $\texttt{fold}(\texttt{l} \cdot \langle\rangle)$, and closed under formation of the successor of any other e of type `nat`, given by $\texttt{fold}(\texttt{r} \cdot e)$. Dually, `conat` is the largest (most permissive) type of expressions e for which the unfolding, $\texttt{unfold}(e)$, is either zero, given by $\texttt{l} \cdot \langle\rangle$, or to the successor of some other e' of type `conat`, given by $\texttt{r} \cdot e'$.

Because `nat` is defined by the composition of its introduction forms and sum injections, it is clear that only finite natural numbers can be constructed in finite time. Because `conat` is defined by the composition of its elimination forms (unfoldings plus case analyses), it is clear that a co-natural number can only be explored to finite depth in finite time—essentially we can only examine some finite number of predecessors of a given co-natural number in a terminating program. Consequently,

1. there is a function $i : \texttt{nat} \to \texttt{conat}$ embedding every finite natural number into the type of possibly infinite natural numbers; and
2. there is an "actually infinite" co-natural number ω that is essentially an infinite composition of successors.

Defining the embedding of `nat` into `conat` is the subject of Exercise **15.1**. The infinite co-natural number ω is defined as follows:

$$\omega \triangleq \texttt{gen}(x.\texttt{r} \cdot x; \langle\rangle).$$

https://doi.org/10.1017/CBO9781316576892.017 Published online by Cambridge University Press

One may check that $\mathtt{unfold}\cdot(\omega) \longmapsto^* \mathtt{r}\cdot\omega$, which means that ω is its own predecessor. The co-natural number ω is larger than any finite natural number in that any finite number of predecessors of ω is non-zero.

Summing up, the mere fact of being a solution to a type equation does not uniquely characterize a type: there can be many different solutions to the same type equation, the natural and the co-natural numbers being good examples of the discrepancy. However, we will show in Part VIII that type equations have unique solutions (up to isomorphism) and that the restriction to polynomial type operators is no longer required. The price we pay for the additional expressive power is that programs are no longer guaranteed to terminate.

15.5 Notes

The language **M** is named after Mendler, on whose work the present treatment is based (Mendler, 1987). Mendler's work is grounded in category theory, specifically the concept of an algebra for a functor (MacLane, 1998; Taylor, 1999). The functorial action of a type constructor (described in Chapter 14) plays a central role. Inductive types are initial algebras and coinductive types are final coalgebras for the functor given by a (polynomial or positive) type operator.

Exercises

15.1. Define a function $i : \mathtt{nat} \to \mathtt{conat}$ that sends every natural number to "itself" in the sense that every finite natural number is sent to its correlate as a co-natural number.
(a) $\mathtt{unfold}\cdot(i(\mathtt{z})) \longmapsto^* \mathtt{l}\cdot\langle\rangle$.
(b) $\mathtt{unfold}\cdot(i(\mathtt{s}(\overline{n}))) \longmapsto^* \mathtt{r}\cdot i(\overline{n})$.

15.2. Derive the iterator, $\mathtt{iter}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$, described in Chapter 9 from the recursor for the inductive type of natural numbers given in Section 15.1.

15.3. Derive the stream generator, $\mathtt{strgen}\, x\, \mathtt{is}\, e\, \mathtt{in} <\mathtt{hd} \hookrightarrow e_1, \mathtt{tl} \hookrightarrow e_2>$ from the generator for the coinductive stream type given in Section 15.1.

15.4. Consider the type $\mathtt{seq} \triangleq \mathtt{nat} \to \mathtt{nat}$ of infinite sequences of natural numbers. Every stream can be turned into a sequence by the following function:

$$\lambda\,(\mathtt{stream} : s)\,\lambda\,(n : \mathtt{nat})\,\mathtt{hd}(\mathtt{iter}\, n\, \{\mathtt{z} \hookrightarrow s \mid \mathtt{s}(x) \hookrightarrow \mathtt{tl}(x)\}).$$

Show that every sequence can be turned into a stream whose nth element is the nth element of the given sequence.

15.5. The type of lists of natural numbers is defined by the following introduction and elimination forms:

$$\frac{}{\Gamma \vdash \mathtt{nil} : \mathtt{natlist}} \tag{15.10a}$$

https://doi.org/10.1017/CBO9781316576892.017 Published online by Cambridge University Press

$$\frac{\Gamma \vdash e_1 : \mathtt{nat} \quad \Gamma \vdash e_2 : \mathtt{natlist}}{\Gamma \vdash \mathtt{cons}(e_1; e_2) : \mathtt{natlist}} \tag{15.10b}$$

$$\frac{\Gamma \vdash e : \mathtt{natlist} \quad \Gamma \vdash e_0 : \tau \quad \Gamma\, x : \mathtt{nat}\, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}\, e\, \{\mathtt{nil} \hookrightarrow e_0 \mid \mathtt{cons}(x; y) \hookrightarrow e_1\} : \tau} \tag{15.10c}$$

The associated dynamics, whether eager or lazy, can be derived from that of the recursor for the type `nat` given in Chapter 9. Give a definition of `natlist` as an inductive type, including the definitions of its associated introduction and elimination forms. Check that they validate the expected dynamics.

15.6. Consider the type `itree` of possibly infinite binary trees with the following introduction and elimination forms:

$$\frac{\Gamma \vdash e : \mathtt{itree}}{\Gamma \vdash \mathtt{view}(e) : (\mathtt{itree} \times \mathtt{itree})\,\mathtt{opt}} \tag{15.11a}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma\, x : \tau \vdash e' : (\tau \times \tau)\,\mathtt{opt}}{\Gamma \vdash \mathtt{itgen}\, x\, \mathtt{is}\, e\, \mathtt{in}\, e' : \mathtt{itree}} \tag{15.11b}$$

Because a possibly infinite tree must be in a state of continual generation, viewing a tree exposes only its top-level structure, an optional pair of possibly infinite trees.[1] If the view is `null`, the tree is empty, and if it is $\mathtt{just}(e_1)e_2$, then it is non-empty, with children given by e_1 and e_2. To generate an infinite tree, choose a type τ of its state of generation, and provide its current state e and a state transformation e' that, when applied to the current state, announces whether or not generation is complete, and, if not, provides the state for each of the children.

(a) Give a precise dynamics for the `itree` operations as just described informally. *Hint*: use generic programming!

(b) Reformulate the type `itree` as a coinductive type, and derive the statics and dynamics of its introduction and elimination forms.

15.7. Exercise **11.5** asked you to define an RS latch as a signal transducer, in which signals are expressed explicitly as functions of time. Here you are asked again to define an RS latch as a signal transducer, but this time with signals expressed as streams of booleans. Under such a representation, time is implicitly represented by the successive elements of the stream. Define an RS latch as a transducer of signals consisting of pairs of booleans.

Note

1 See Chapter 11 for the definition of option types.

https://doi.org/10.1017/CBO9781316576892.017 Published online by Cambridge University Press